

Implementing CRCs

At an unrecorded event lost in the mists of time, a caveman named Og sent a message proposing peace, via smoke signal, to the signal officer of the neighboring tribe. Unfortunately, a gust of wind disrupted the pattern of a particularly important part of the message, and the message was misunderstood. Sadly, Og's tribe was wiped out.

Ever since then, people have been sending messages by methods ranging from smoke signals to laser beams to spread-spectrum communications satellites, and messages have been arriving garbled, with more or less disastrous results. It's not surprising, then, that people have long sought ways to make sure that their messages were received properly. Lots of methods have been tried over the years.

Today, one of the most effective and popular error-detection methods is the cyclic redundancy check (CRC). It's used in virtually every field where transmitting serial data is involved. It's even built into your disk-drive controller. Most of you have already heard of the CRC, and you've almost certainly used it, whether you are aware of it or not.

Though ubiquitous, the CRC algorithm is surrounded by an inordinate amount of fog. We can't even seem to agree on what the acronym stands for (some say it's cyclic redundancy *code*). Ask hardware people how it works, and they're likely to say, "It's done with shift registers and exclusive-OR gates." Ask mathematicians, and they'll say, "It's done using a polynomial division, modulo two, with the generating polynomial chosen from a closed Galois field." (Oh. Thank you very much. That clears everything right up.) Ask *honest* software types, and they'll tell you, "Search me how it works. I just copied it from the XMODEM code."

In this article, I'm going to try to cut through some of that obscuring fog.

Most of you have already heard of the CRC, and you've almost certainly used it, whether you are aware of it or not.

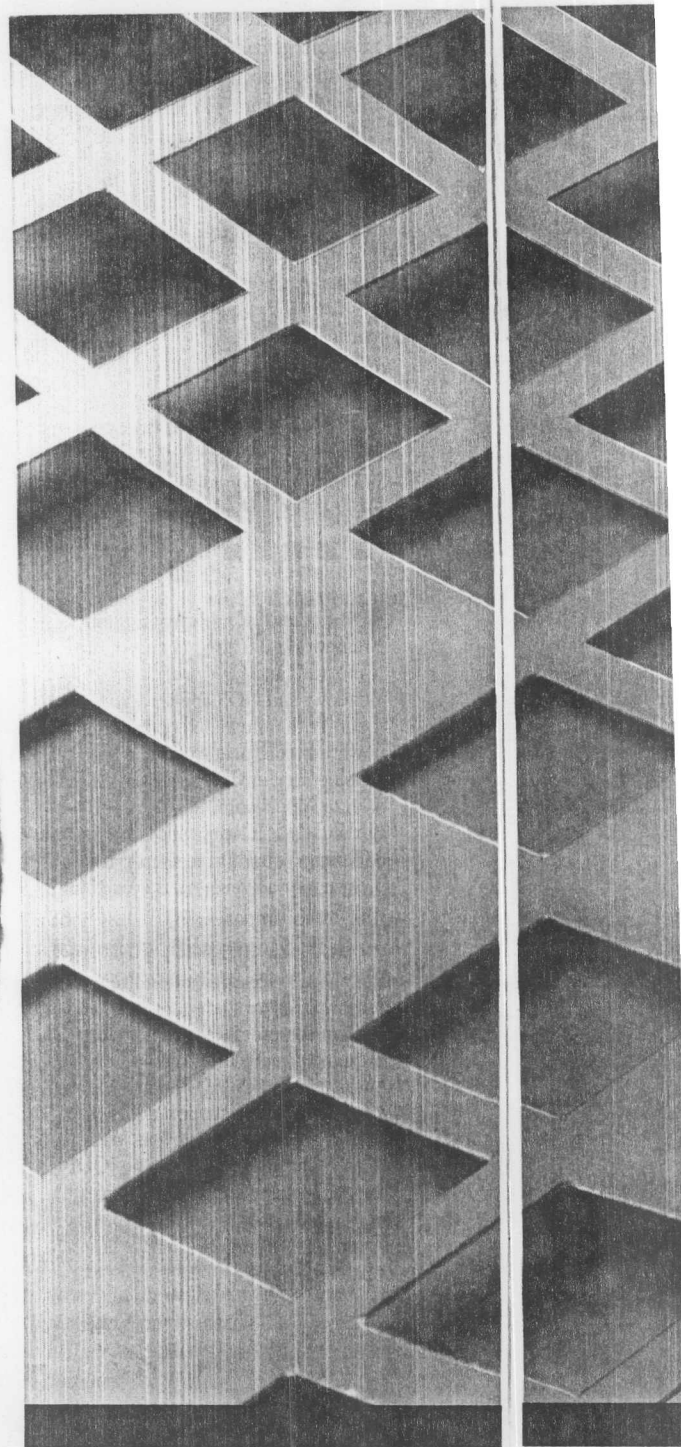
We'll begin with some background as to the motivation and power of the CRC, then continue on to the gory details. I'll spend just enough time discussing polynomials to convince you that they're not needed, and then we'll get on to some efficient code implementations. I'll also discuss some of the gotchas you should beware of.

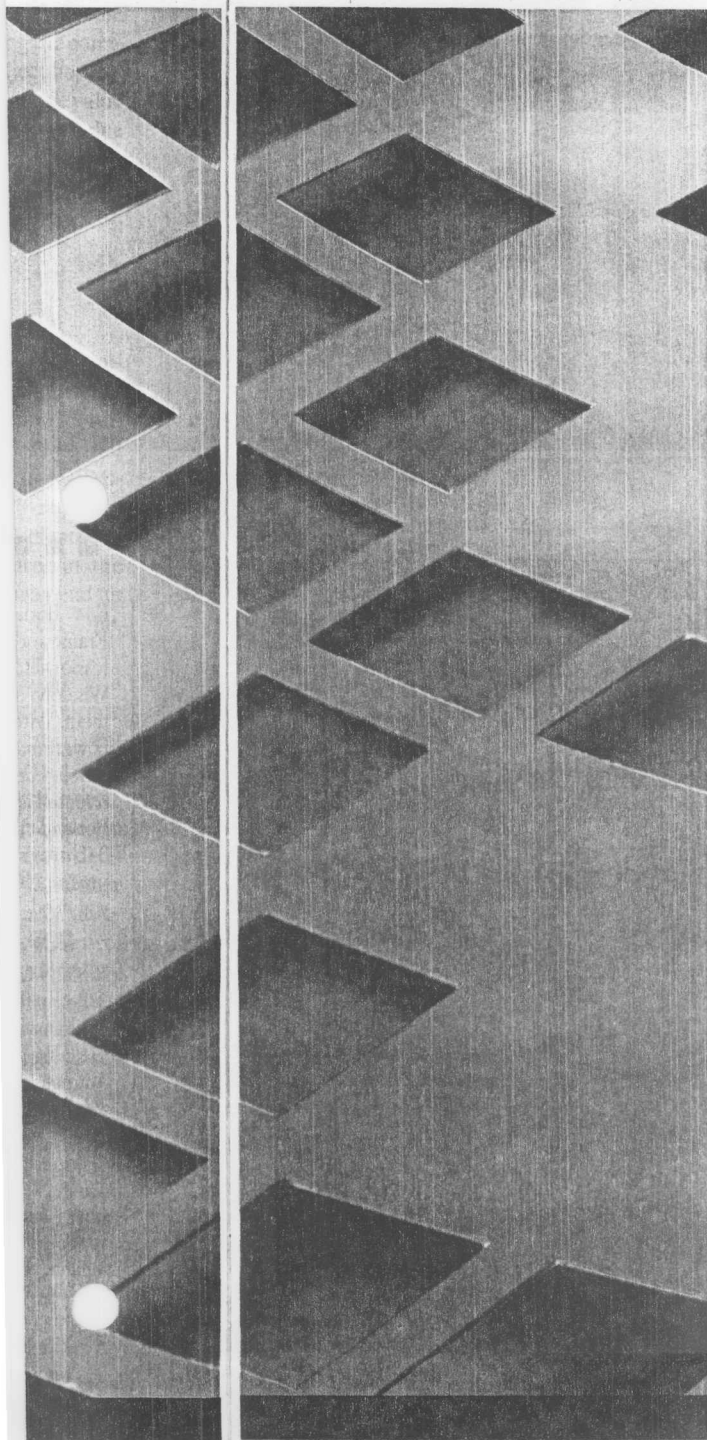
With any kind of luck and some persistence on your part, you'll leave this article with a real understanding of how and why the CRC works. At the very least, you should be able to say, "Search me. I just copied it from Crenshaw."

SOME BACKGROUND

Ever since the landmark Og affair, people have been seeking ways to make sure that their messages get across without being misunderstood. In today's information age, that's more important than ever. We can accomplish this goal in many ways, but they all boil down to these three requirements:

- Along with the message, provide the recipient with some way to know if it got there correctly.
- The recipient should send a return message, acknowledging receipt or asking for a retry.





■ Continue to send the message until it gets to its destination ungarbled.

Effective data transmission involves two aspects: error detection and error transmission. The CRC is an error *detection* mechanism. Yes, I know that some error detection and correction methods such as the Hamming codes exist, but today, almost everyone still separates the two aspects and counts on retransmission for the correction half of the system.

How can the recipient of a message tell whether it's been garbled? The most obvious way is to send every message twice. But that method involves a lot of overhead—at least double the transmission time; much more if the error rate is high enough so that errors are likely. And that's bad. Time is money.

We can make things a little better by splitting the message up into smaller blocks, and seeking confirmation on each block. This way, we don't have to retransmit the whole message, but only the blocks that were garbled. But we're still limited to a minimum of twice the transmission time. What we really need is a method to test the message for correctness, without using many bits of the message.

Of course, the ancient kings knew how to authenticate messages. They used a courier and sealed the messages with their private seal, carefully monogrammed to discourage forgery. No one could tamper with the message without breaking the seal. We can't do that with messages that are sent electronically, but we can steal the general idea. You append some kind of code to the message that's virtually certain to be altered if the message is tampered with or otherwise garbled.

Early on, someone got the idea to use a code (sometimes called the block-check code, or BCC) that could somehow be computed from the body of the message. The recipients apply their

magic decoder rings to the body of the message, and compare the result to the code that was received. If they don't match, the message must be in error. To be efficient, the code should be small compared to the message body. Of course, the smaller the code, the more the possibility of aliasing, where two different messages can generate the same code. But with modern methods such as the CRC, this scenario is extremely unlikely, as we shall see.

The smallest code is a single bit. That's the idea behind the parity bit, which is a measure of the number of ones in the message's binary representation—zero for an even number, one for an odd number. Since it's short, it's also easy to get aliasing, so parity bits only work for very small message blocks—typically one byte.

Another choice is the checksum. It's a popular one, because it's so easy to understand and compute. We simply add all the bytes in the message as though they were binary numbers (ignoring carries). Typically, the resulting sum is appended to the message as its twos complement, so that the recipient, adding the same characters plus the checksum, should get zero. Intel hex and Motorola S-record formats use either a 1- or 2-byte checksum.

Finally, we have the CRC. As nearly as I can tell, it arrived on the scene in 1961. The idea behind it is *exactly* the same as for the checksum: compute a 1- to 4-byte BCC, calculated from the message body, and append it to the message. The only difference is the algorithm embodied in the magic decoder ring.

WHY USE A CRC?

A little reflection should convince you that the reliability of any error-detection mechanism (due to aliasing) is a function of the size of the appended code. An N-bit code can only have 2^N unique values. A

Implementing CRCs

single parity bit can only detect single (or any odd number of) bit errors. If the likelihood of multiple errors is low, it works fine. But if it's high, there's only a 50% chance that a parity bit will reflect the error.

Similarly, a checksum or CRC of eight bits (one byte) can have only 256 possible values, which means that, on a statistical basis, we have a 0.4% chance that a bogus message will slip through. Inversely, the reliability of the error-detection mechanism is 99.6%. For two bytes, the value is 99.9985%. Now, if the CRC also uses two bytes, you might think that it can't possibly be better than a checksum. So why bother with the more complex algorithm?

That assumption would be true if transmission errors were truly random, but they're not. In the real world, two kinds of errors typically occur: random-bit errors and burst errors (in which several adjacent bits are garbled). The CRC shines in catching these two errors. Given the proper choice of algorithm, a 16-bit CRC can detect:

- 100% of all single-bit errors
- 100% of all two-bit errors
- 100% of all odd numbers of errors
- 100% of all burst errors less than 17 bits wide
- 99.9969% of all bursts 17 bits wide
- 99.9985% of all bursts wider than 17 bits (the same as a checksum)

For extreme reliability, using 32 bits improves this performance even more, by five orders of magnitude! Those statistics are pretty impressive, which is what makes the CRC so popular.

The nature of the checksum is such that errors tend to be localized, sometimes changing only a single bit. Offsetting or systematic errors (say, a stuck bit on the UART) can often be missed.

By contrast, any error in the message quickly propagates into all the bits of

The CRC is equally valid for small and large messages, which is why it is so popular and considered worth the extra computation required.

the BCC, so the probability of two offsetting errors is virtually nil. In that sense, the CRC for a message is a bit like a hologram of a scene: the bits of the data are scattered through all the bits of the BCC. For the same reason, the CRC is equally valid for small messages as for large ones. *That's* the reason the CRC method is so popular and considered worth the extra computation required.

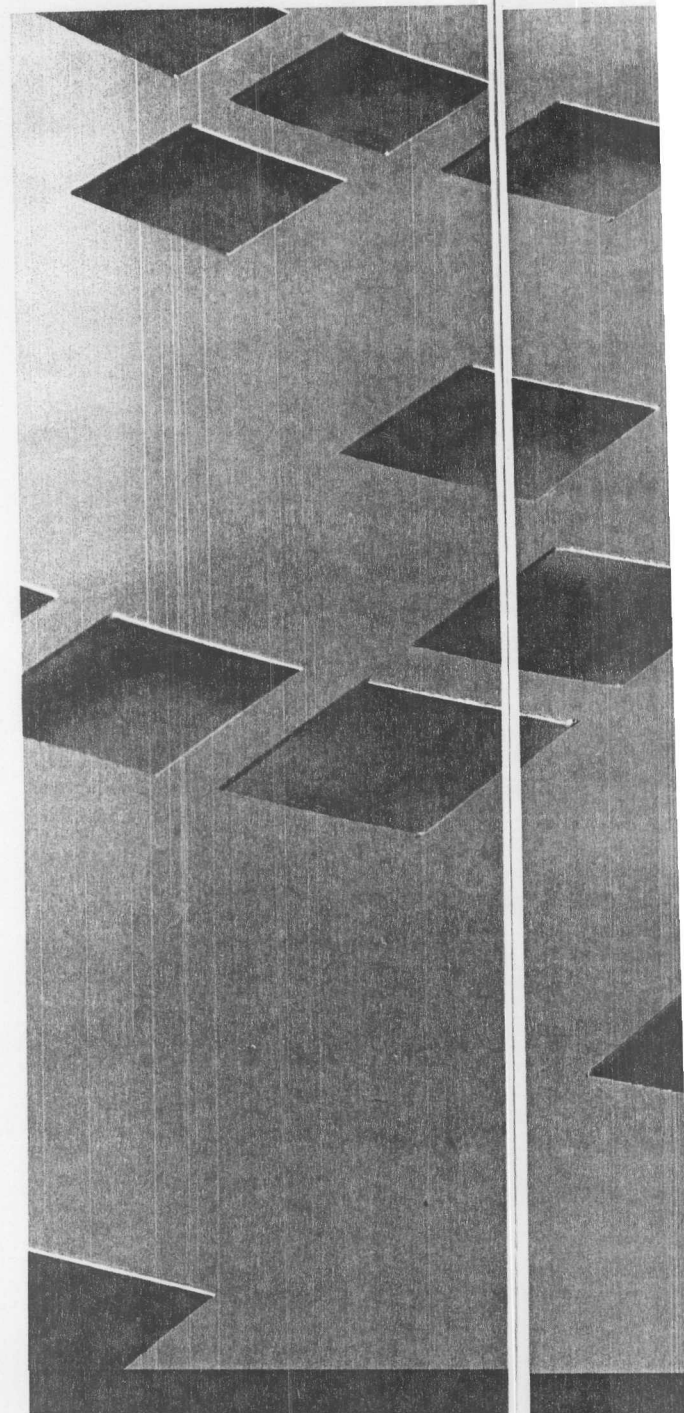
CLEARING THE FOG

When I first asked someone to explain the CRC to me, I was told, "It's the remainder of a polynomial division, modulo two." I was then given the generating polynomial:

$$x^{16} + x^{15} + x^{13} + x^7 + x^4 + x^2 + x + 1.$$

Huh? What the heck does that mean? And what is x , anyway? A data bit? Shall I give it a value?

A more meaningful way to understand what's going on with CRCs must exist. It's easier if we first digress into the related field of pseudo-random



numbers. Let's play a little game, of the kind often seen in I.Q. tests or math puzzle books. Look at this a sequence of numbers:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14

Quick! What's the next number in the sequence?

If you didn't say 15, please move on to the next article. This one's not for you. For the rest of you, please give me the next number in this sequence:

1, 9, 13, 15, 14, 7, 10, 5, 11, 12, 6, 3, 8, 4, 2

Not so easy, is it? (The answer is one.) It's not *supposed* to be easy, because the second sequence is a set of what are called pseudo-random numbers. The "pseudo" part is because they are not really random. The sequence will repeat over and over forever. But by every *other* measure of randomness you care to apply (average, standard deviation, and so on), the sequence is random. Hence the name. The sequence covers all the 4-bit integers, except zero.

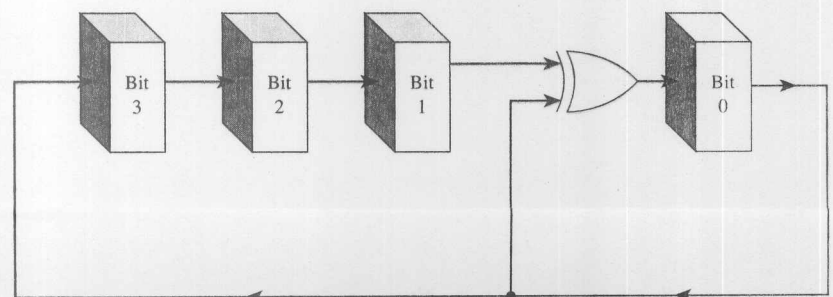
How did we get this sequence? In this case, the magic decoder ring is a hardware circuit, as shown in Figure 1.

It is a shift register, with one exclusive-OR gate added. Place any number in the sequence into the bits of this register, mentally cycle it, and you will soon convince yourself that it does indeed generate the 15-number sequence. It's a source of pseudo-random numbers, better known as white noise, and you'll find one in every Nintendo game. Of course, in the real world, these generators use more than four bits. In general, an N-bit pseudo-random number generator can generate $2^N - 1$ unique numbers before repeating. The -1 is there because we can't allow the value zero. If the generator ever gets all zero bits, no feedback can occur, so it will remain at zero. Unfortunately, not just any old set of feedback taps (the exclusive-OR gates) will do. For example, if the gate were at Bit 1 instead of Bit 0, we would end up with three shorter sequences:

- a. 1, 10, 5, 8, 4, 2
- b. 3, 11, 15, 13, 12, 6
- c. 7, 9, 14

Now, you begin to see the tricky part about this process. We want a *maximal-length* sequence, and to do get it, the taps must be carefully chosen. And *that's* where the polynomials come in.

Figure 1
A hardware circuit.



Implementing CRCs

POLY-CONFUSED

My foray into the theory behind all this stuff led me deep into the bowels of group theory. I learned more than I care to know about Abelian groups and Galois fields. I sure won't burden you with all that stuff. You need only know that it's this theory that leads us into polynomials and polynomial division. It turns out that, for a maximal-length pseudo-random number sequence, you need a polynomial that's irreducible (one that cannot be factored). It's the polynomial equivalent of a prime number. Finding them is not easy. It's as hard as finding a large prime number, and complicated by the harder division process, but the

For a maximal-length pseudo-random number sequence, you need a polynomial that's irreducible. It's the polynomial equivalent of a prime number.

concept is straightforward.

In the normal world of polynomials (not modulo two), the polynomial:

$$x^2 + 2x + 1$$

is reducible, because it can be factored

into:

$$(x + 1)(x + 1).$$

Similarly, $x^2 - 1$ can be factored into:

$$(x + 1)(x - 1).$$

We look for factors of polynomials the same way as with integers: by trial and error, using division. We learned polynomial division (sometimes called synthetic division) in algebra class (remember?). It's much like ordinary division, except that you have to write the polynomials with the highest order first, and use only the first terms in the divisor and dividend. If, for example, the highest order term in the dividend is x^3 and the divisor is x , the first term in the quotient has to be x^2 . From there, it's much like ordinary division: multiply, subtract, and bring down the remainder. If the remainder is zero, the dividend is reducible.

The polynomial arithmetic used for CRC theory is a little different. It uses modulo-two arithmetic. The reason is

CROSS DEVELOPMENT SYSTEMS

Ansi-C/Modula-2
Hitachi 8/500

68000 Family

C Development
68HC11

compatible...
Windows™ 3.0

Open-Look™
Motif™



Software Environments
PO Box 2129
Mission Viejo, Ca. 92690
(800) 927-1481

CIRCLE # 11 ON READER SERVICE CARD

We're bla
control s
performa
efficiency
Introd
RISC pro
Division
from the
embedd
Our fir
MB8693
eration o
designed
— for mu
competi
speeds u

TAIPEI

Implementing CRCs

simple: ordinary arithmetic doesn't permit the group properties that the mathematicians need for their powerful methods to work.

In modulo-two arithmetic, we don't have carries or borrows from any bit positions. The rules for adding two bits are:

$0 + 0 = 0$
 $0 + 1 = 1$
 $1 + 0 = 1$
 $1 + 1 = 0$

Look familiar? It is also the truth table for a simple half-adder, or exclusive-OR. As it turns out, subtraction gives the same rules, so in this funny world of

modulo two, addition and subtraction are the same.

Already, we've cleared out one fog factor. The next time someone says "modulo-two arithmetic," just think "exclusive OR." Simple. Also, in this world twos and threes don't exist ($1 + 1$

$= 0$, not 2), so the only coefficients polynomials can have are zero or one. Polynomial division in modulo two is as easy as it is in ordinary arithmetic, once you get used to the new rules. An example is shown in Figure 2.

It looks straightforward, but writing

Figure 2
Modulo 2 division.

$$\begin{array}{r}
 x^2 + x + 1 \overline{) x^4 + x^3 + x^2 + 1} \\
 \underline{x^4 + x^3 + x^2} \\
 x^3 + x^2 + x \\
 \underline{x^3 + x^2 + x} \\
 x^2 + x + 1 \\
 \underline{x^2 + x + 1} \\
 \text{(remainder) } 0
 \end{array}$$

A zero remainder means the dividend is factorable:

$$x^4 + x^2 + 1 = (x^2 + x + 1)(x^2 + x + 1)$$

The Toughest 8051 Jobs Just Got Easier!

"The challenge for 8051 designers is to squeeze all the performance out of the 8051 at the lowest cost, and Franklin is there with "C" tools that help you meet it."

Franklin Introduces 8051 "C" Professional Developers Kit!

Handle Bigger Jobs.

Our *Best in the Industry* "C" compiler now supports multi-tasking and bank switching with up to 16 banks of memory for code and data. No other *bank management* scheme is as code and execution time efficient.

The Fastest Just Got Faster.

With all its new features, the C-51 Compiler is still the fastest, most code efficient "C" compiler available. And for big jobs, C-51 is faster than ever for both compile and execute times!

Debugging Just Got Easier.

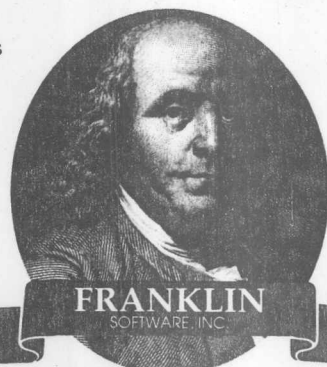
Use DScope-51's powerful symbolic debugging facilities to control simulation, or remote target debugging with Monitor-1, or with popular emulators. The same interface manages all phases of debugging!

Introducing RTX-51 Operating System!

It's the first real time multi-tasker designed specially for the 8051. RTX-51 is very small, as little as 800 bytes for code, so it works even on single chip versions of the 8051. And task switches are very fast! With a full set of OS features, and support for all 8051 variants, RTX-51 opens up new vistas for 8051 development!

Move up to the Professional Developers Kit!

Call (408) 296-8051 or FAX us at (408) 296-8061 for your FREE evaluation package!



	FRANKLIN PROFESSIONAL	BSO/TASKING	IAR/Archimedes ICC8051
Total Execution Time (large model)	6.744 sec 6.287 sec	6.753 10.98	8.386 17.295
Module Code Size (large model)	188 bytes	266	238
dynamic Data Size	188 bytes	303	343
Total Code Size (large model)	41 bytes	126	96
	1223 bytes	1936	1626
	1292 bytes	3467	2088
Whetstones/Second	13	Float not yet supported	3
Dhrystones/Second	203	163	90

Franklin Software, Inc.

888 Saratoga Ave. #2, San Jose, CA 95129

CIRCLE # 13 ON READER SERVICE CARD

Here's enlighten platforms for ind applications.

There's a platf combines the adv industry leaders - and Intel.

Intel is well-kno embedded control microprocessors. performance, desi quality represent And among Intel's products for indus the new 302i rack-range of Intel386[™] platforms, and sof such as iRMX[®] fo

Hamilton/Avnet c systems engineerin our extensive value capabilities.

Put them all toget result is superior sy

Hamilton/Avnet a building a reputati So see the differenc For details, call Ha free, 1 (800) 442-64



Intel386 and i486 are trade is a registered trademark o

* Windows is a trademark o CORP.

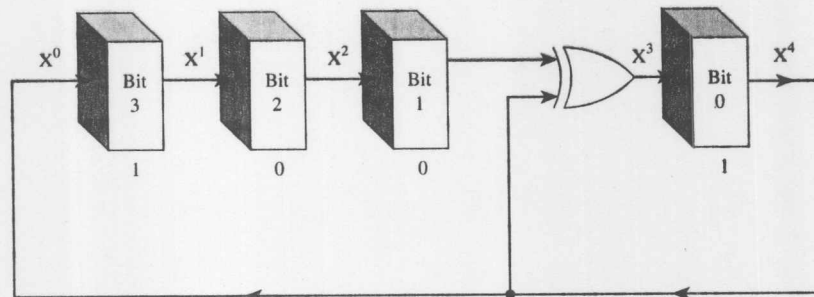
Implementing CRCs

Figure 3
The easy way.

```

      111
111 ) 10101
      111
      ---
      100
       111
       ---
        111
        111
        ---
         0
    
```

Figure 4
Three representations of a hardware circuit.



Generating Polynomial = $x^4 + x^3 + 1$
Feedback factor = 1001b

down all those polynomial terms gets tedious, especially if we're talking about 16th-order polynomials! Fortunately, once I'd done this problem a few times, I finally figured out that it's not necessary to write the x s down at all—just the coefficients. Figure 3 shows the same division in this simplified

form. Looks a lot easier, doesn't it? In fact, it's simply binary division, but using exclusive ORs instead of subtraction. Since it's binary, of course, you don't really need to perform a multiplication. At each step, we either subtract the divisor or we don't. We don't need to compare the two terms (we don't worry if one polynomial is bigger than the other. Since addition and subtraction are the same, the concept of "bigger" does not exist). We only have to look at the leading terms. If they have the same order, we subtract.

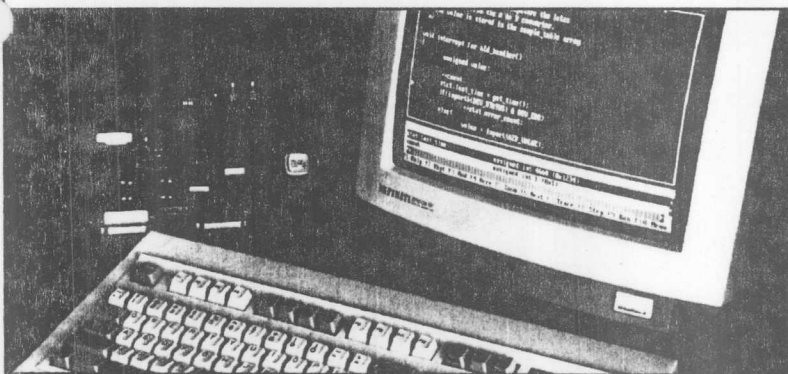
Now, we get rid of the second fog factor. From now on, when someone starts talking to you about polynomial division, you will know that what they really mean is ordinary binary division—modulo two.

In Figure 3, each product to be subtracted is written down one place to the right, which is equivalent to shifting the dividend to the left. So, what we're really talking about is shifting, then subtracting (or exclusive OR-ing) a constant value. But that's exactly what's going on in Figure 1. The operations are entirely equivalent.

We only have a couple of annoying, but minor, complications. First, the shift register shifts right, while the division operation shifts the dividend left. We'll just have to remember to reverse the bit order.

Second, the division subtracts first, and then shifts for the next round. In the shift register, we look at the least significant bit (LSB) shift, then perform the

On Target with Turbo Debug!



Place your Microsoft or Borland C Code in the target system and debug it as it runs with Datalight's **C_thru_ROM**...

C_thru_ROM eases ROM development with all versions of Borland-C, C++, and Microsoft C. New features include Turbo Debug support and remote debugging via a ROM socket. **C_thru_ROM** is a complete ROM development package that works with your choice of compiler to quickly get your application running in ROM. Includes: full 80x86 locator which outputs in Intel OMF, hex, and binary; ROMable startup code; remote debugging via Turbo remote debugger or CodeView-like debugger; full floating point support; ROMable library (*printf*, *malloc*, etc.); and much more. Call, write, or fax today for full details.

Free Demo Disk! Call Today Toll-Free 1-800-221-6630

Datalight

17455 68th Ave NE, Suite 304, Bothell, WA 98011, USA • 206-486-8086 • fax 206-486-0253

CIRCLE # 15 ON READER SERVICE CARD

Yes,

I would I

I am curr

☐ Add m

Name

Company

Address

Phone

If you have

© 1990 Applied

N

You
comm
solving
embed
problem

Hu
money
needs n
worked
scale p
it often
of the r
today's
and 32-

At A
we're h
hardwa
better e

In the U.S. and C
44-(0)-296-62546
fact Applied Micro
registered tradema

Implementing CRCs

exclusive OR. The divisor is shifted one bit relative to its equivalent polynomial.

It may not seem so, but at this point the fog is almost completely dissipated. We now have a "rosetta stone" that lets us translate between the three representations. Consider, for example, the polynomial:

$$x^4 + x^3 + 1.$$

In the binary form needed to do a division, this polynomial is 11001. The number has a one bit for every non-zero term, starting with the highest order (the highest and lowest bits will always be ones, by the way, for any useful polynomial.)

You might well ask why I even bother mentioning polynomials, with their awkward representation and built-in confusion factor.

To get the equivalent shift register, we first turn the number around, then shift it right one place, to get 1001. This number is the feedback factor that drives the generator. Place a tap at the entrance to every bit that has a one in its feedback factor.

Alternatively, just remember that the left-hand bit of the shift register cor-

responds to the *lowest-order* term in the polynomial; that is, x^0 or one. The three representations are shown together in Figure 4 (p.26). I advise you to cherish this figure. You will rarely see all three representations identified on the same figure, so it is literally a rosetta stone. With it, you need never be confused by polynomials again.

You might well ask why I even bother mentioning polynomials, with their awkward representation and built-in confusion factor. Good question. I wish I had a good answer. Tradition seems to be the only reason. The polynomial representation is good if you have to deal with the field theory to indentify a good polynomial. For the rest of us, it's completely unnecessary, and only generates fog. Give me the feedback factor, or better yet, the whole equivalent circuit. But don't be surprised when others give you the polynomial form, because that's the way CRCs are normally defined.

ON TO THE CRC

Of course, the circuit shown in Figure 1 is not a CRC generator, just a pseudo-random number generator. It cycles constantly, unchanged, because it doesn't receive input. That's easy to fix, though.

Remember that our pseudo-random number generator cannot deal with zero? That bothers some people, since clearing all the bits of a shift register is a common thing to do. The problem is easily fixed by inverting the output, which will still give us a maximal-length sequence, just a different one:

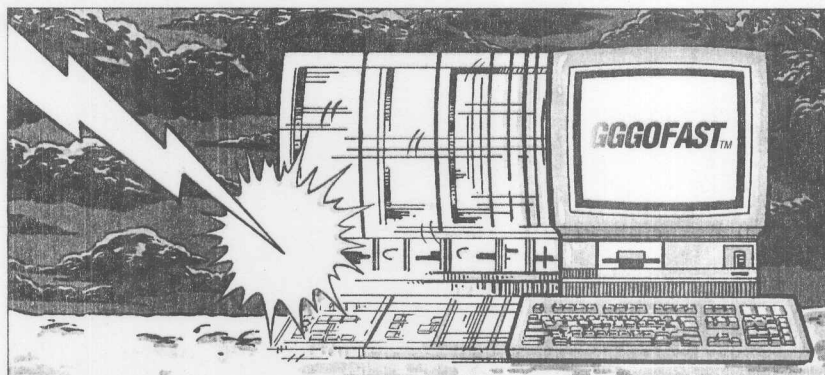
0, 9, 4, 11, 5, 2, 8, 13, 6, 10, 12, 15, 7, 3, 1

Now we can have a zero, but not a 14. No help for that.

Next, let's get fancy. Suppose we replace the inverter at the output by one more exclusive-OR gate (the missing one at x^4), and let the data diddle it high and low. Boy, *that* should really confuse things!

But that's precisely the concept behind the CRC algorithm. The idea is to get the effect of the data bits widely scattered throughout the BCC, which is what all those taps do.

With that long buildup, we're now poised to look at true CRCs in all their glory. The most popular standards are



GOFAST™

Lightning-Fast Floating Point Accelerators

Empower your 80x86 applications with GOFAST accelerators. Fast, reentrant, and ROMable, GOFAST accelerators boost performance and make sure you can embed your application.

Link and go with C: Microsoft®, Borland®, Intel®, MetaWare®, and WATCOM®. Dynamically replace 80x87 coprocessors during execution.

GOFAST IEEE accelerators are optimized for 8051, 8096, 8086, 80386, i960, 6801, 6809, 68HC11, 68xxx, 8085, Z80, R3000 and more.

Call for your free GOFAST information diskette today: 503-641-8446; FAX 503-644-2413; 800-356-7097.



14215 NW Science Park Drive
Portland, OR 97229

U S SOFTWARE®

© 1991 US Software Corporation. GOFAST and DynaCOP are trademarks of US Software Corporation. All other trademarks belong to their respective owners.

CIRCLE # 17 ON READER SERVICE CARD

Speed
with M
for 17

Fast pro
key to s
tronics.
your pro
NEC's 17
control
SIMPLE
your mo
grammi

The r
range o
specific
end dig
remote
tions ar
SIMPLE

For fas

Tel: 1-800-
Tel: 0-
Tel: 0908-
Tel: 02-

shown in Figure 5 (p.43). The CCITT algorithm is the international standard. The CRC-16 is most often used in the United States. The CRC-8 is pretty

Listing 1

The Serial CRC program.

```
uses HexOut;

const Feedback = $8408; { CRC-CCITT }
               $A001; { CRC-16 }
               $8000; { LRCC-16 }

var CRC: Word;

{ Process the CRC for a Single Byte }

Procedure UpdateCRC(B: Byte);
var i: Integer;
begin
  for i := 1 to 8 do begin
    if Odd(B) Xor Odd(CRC) then
      CRC := (CRC Shr 1) Xor Feedback
    else
      CRC := CRC Shr 1;
    B := B Shr 1;
  end;
end;

{ Send a Message String }

Procedure SendString(S: String);
var i: Integer;
begin
  for i := 1 to Length(S) do
    UpdateCRC(Ord(S[i]));
end;

{ Send Message, Including CRC Bytes }

Procedure SendMessage(S: String);
var OldCRC: Word;
begin
  CRC := 0;
  SendString(S);
  OldCRC := CRC;
  WriteLn('CRC Computed is ');
  WordLn(CRC);
  UpdateCRC(OldCRC);
  UpdateCRC(OldCRC Shr 8);
end;

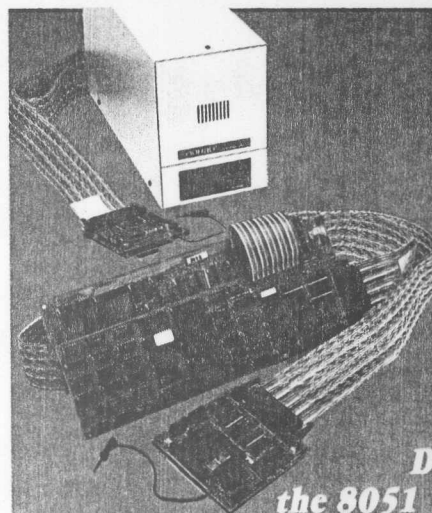
begin
  SendMessage('THE, QUICK, BROWN, FOX, 0123456789');
  WordLn(CRC);
  ReadLn;
end.
```

much obsolete, and the CRC-12 is only used when bytes are six bits.

As you can see, real CRCs differ from our toy one only in the number of bits—the concept is the same. For the record, however, the criteria for the generating polynomial (feedback factor, for those of us in the know) is different. Rather surprisingly, the CRC generators do not generate maximal-length sequences. In fact, the polynomials are deliberately chosen to be reducible by the factor $x + 1$, because that happens to eliminate all odd-bit errors. The fact is, the polynomials are carefully tailored

to get optimal error-detection out of the system. Don't worry about how they do it. Better than us.

I should mention a few points about how the data gets handled. From a mathematician's point of view, the entire message is considered to be a string of bits, each bit corresponding to a term in a humongous polynomial. So, from their point of view, we are dividing one long polynomial by another one. The BCC is the remainder. From our point of view, we are dividing two binary numbers. In the real world, though, we know we're really sending messages a byte at



8051 & 68HC11

PC-Based
In-Circuit Emulators

Nohau
Covers All Your
Development Needs for
the 8051 and 68HC11 Families!

Free Demo

You can start your debugging with this **FREE** demo simulator. You can load up to 512 bytes of code, assembler, C, or PL/M and do full debugging/simulation in assembly and source level. A great way to get started for **FREE**. Fantastic for schools! Just call and we'll send it!

Full Simulator

The full-blown simulator is an extension of the DEMO. You can load up to 64K of code and use 64K of XDATA space. You can program an "external environment" to interact with your code to simulate your target system. The emulator is the hardware extension of the simulator!

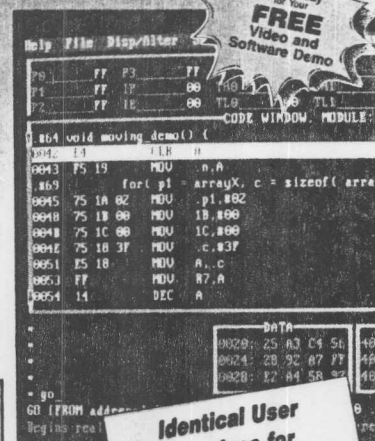
In-Circuit Emulation

The 30MHz real-time emulator has been the industry standard for years. With its complex breakpoint logic and advanced trace, nobody can beat it for performance. Plug-in or RS-232 configuration. All 8051 derivatives are supported!

NOHAU
CORPORATION

51 E. Campbell Avenue, Campbell, CA 95008
(408) 866-1820 • FAX (408) 378-7869

Call Nohau's 24-hour
information center to
receive info on your
FAX 408-378-2912



**Identical User
Interface for
All Three Products —
You Can't Go Wrong!**

Australia (02) 654 1873, Austria (0222) 38 76 38, Benelux +31 1858-16133, Canada (514) 689-5889, Czechoslovakia 0202-2683, Denmark (42) 65 81 11, Finland 90-452 1255, France (01) 69 41 28 01, Germany 08131-25083, Great Britain 0962-73 31 40, Greece 01-862-9901, Hungary (1) 117 6576, Israel (03) 48 48 32, Italy (011) 771 00 10, Korea (02) 784 784 1, New Zealand (09) 392-464, Portugal 01-80 9518, Norway 02-649050, Singapore (065) 284-6077, Spain (93) 217 2340, Sweden 040-9224 25, Switzerland (01) 740 41 05, Taiwan (02) 7640215, Thailand (02) 281-9596, Yugoslavia 061 621066.

CIRCLE # 20 ON READER SERVICE CARD

a time, and the standard convention for all data transmission is to send the *least* significant bit (LSB) first. It makes sense, then, to think of the LSB of the first byte as the most significant bit (MSB) of the long binary number that we're going to divide (or, equivalently, the highest-order term in the polynomial). That's the reason for the relationship between bit patterns and polynomial terms. It's confusing, because not only are these two items in opposite orders, but if we want to perform the division by hand (as a sanity check, for example), we have to invert the bits of every byte transmitted.

To drive home that point, let's actually perform such a division. The process is shown in Figure 6 (p.40). I've used a single-byte message, the ASCII character M. We'll use the CCITT code. To perform the division, we must append 16 zeros to the message. After all, the dividend has to be larger than the divisor! This step is *not* required when we do the CRC in hardware (or software). The circuitry takes care of that.

Please follow along with this exercise. I know it's painful, but it will remove all the mystery of the process. The polynomial is 17 bits long, so it leaves a remainder of 16 bits, as needed. We don't have to check to see if one term is larger than the other. Since addition is the same as subtraction in this number system, the concept of "larger," except as it refers to the number of bits, does not exist. We look only at the leading bit of each number, lining them up so they'll cancel, which corresponds exactly to the hardware implementation, where the exclusive OR is done strictly on the basis of the two leading (rightmost) bits.

GETTING THE MESSAGE ACROSS

Once we have the BCC, what do we do with it? The answer, of course, is to transmit it, since that's the mechanism the recipient uses to make sure the message is O.K.

Remember that when a checksum (rather than CRC) is used, we usually transmit its twos complement, so the final sum is zero for an error-free message. The CRC works in a similar way, though we don't even have to complement. Suppose that we've sent a mes-

sage that the recipient has received. At this point, the contents of our BCC registers should match. Now, we have to

send the BCC. As the LSB of the BCC arrives at the input, it should match the bit in the recipient's register, so the ex-

Table 1
Test-case data.

Data string	CCITT	SDLC	CRC16	XMODEM
'M'	99E1	9666	35C0	9969
'T'	14A1	1B26	FF01	1A71
'THE'	7D8D	44BE	23B6	1E0A
'THE, QUICK, BROWN, FOX, 0123456789'	7DC5	DF91	B96E	0498

68331, 68332, 68340 Developers: Don't Ignore Background Mode!

If you're ignoring Background Mode, you're ignoring price/performance, ease-of-use, and non-intrusiveness in your debugging system.

Motorola Doesn't.

Motorola realized the benefits of on-chip debugging. That's why they designed Background Mode into every CPU32 microcontroller.

Introducing the EST Series 300!

The EST Series 300 controls your CPU32 target exclusively through a 5 MHz Background Mode link. It's completely non-intrusive, since it doesn't require your target's RAM, ROM, interrupts, or serial port.

Packed with Debugging Power.

Of course, the EST series 300 delivers the features you need: simple and conditional breakpoints on RAM/ROM, data breakpoints, stepping, tracing, view/modify registers and memory, and system diagnostics, at a fraction of the price of a full-blown ICE.

Real-time Simulation.

To get a head start on your software, the EST Series 300 builds in a CPU32 controller and 128K of simulation RAM. Or, our Background Mode system works with your EVS board, right out of the box.

C Source Level Debugging.

EST has ported Intermetrics' XDB 5.0 to the Series 300. XDB delivers a proven windowed interface, complete source and symbolic information, C level tracing, and powerful macros.

Try Background Mode.

Just call us, and we'll send you an EST Series 300 for a *free* 14 day trial. No obligation. No Risk.

(617) 828-5588

Embedded
Support
Tools Corp.

10 Elmwood St., Canton, MA 02021



CIRCLE # 22 ON READER SERVICE CARD

Implementing CRCs

clusive OR is zero. Nothing gets fed back during the shift. If no errors have occurred, the same will be true for every succeeding bit. No data is ever fed back, so the BCC just gets shifted out, leaving a zero value. In this case, zero is the winning score!

PROGRAMMERS DO IT IN SOFTWARE

Until now, I've emphasized the hardware aspects of the CRC generator. As a matter of fact, that's the way the CRC was originally used. Today, it's very often done in software, which will be our focus for the remainder of this article.

A straightforward translation of the algorithm to software is shown in List-

We're not really sending the message anywhere, so I added a little debug output to give the value of the computed CRC.

ing 1 (p.31), written in Turbo Pascal. The bit-by-bit update of the shift register is done verbatim by procedure UpdateCRC. As you can see, it's pretty simple stuff. Following the circuit exactly, we exclusive OR the low-order bits of the BCC and data, and use the result to decide whether or not to feed back into the taps. The only difference between these

16-bit CRCs is the tap positions, which are embodied in the variable FeedBack.

The SendString procedure sends a string of bytes. The SendMessage procedure initializes the BCC and sends its final value. We have to save the value, since it will continue to be updated as its value is sent.

We're not really sending the message anywhere, so I added a little debug output to give the value of the computed CRC. The final value printed should always be zero.

By the way, one of the things sorely needed in working with this algorithm is some test-case data. Fortunately, Greg Morse was kind enough to give us some test cases, which I've supplemented a bit in Table 1 (p.33). They're an invaluable aid for debugging.

A TABLE-DRIVEN CRC

The verbatim, bit-by-bit algorithm gets the job done and is fine for most applications. But it turns out that we have a better, and much faster, algorithm. Since we're really sending stuff in bytes rather than bits, wouldn't it be nice if we could generate the new BCC for a byte in one step? Because of the MixMaster nature of the CRC algorithm, it's certainly not

Multitasking or not
Byte-BOS™
is a must for your real-time
embedded or PC software applications.

Not sure about multitasking? Even if your application uses just one task, Byte-BOS can offer you functions for setting up periodic sampling, driving state machines from interrupts, handling serial I/O for "on board" and external UARTS, fast memory allocation, low power handling and more.

Thinking of writing your own multitasking kernel? For about a week's pay for a software engineer, Byte-BOS provides you with no royalty C source code and thorough documentation for a real-time preemptive multitasking kernel with a robust set of supporting functions.

Looking to buy a multitasking kernel? Byte-BOS saves you time with integration. All versions of Byte-BOS are tested and configured for specific C compilers and Byte-BOS is delivered with a working application configured to a host of hardware targets.

Using more than one processor? Byte-BOS is designed to be portable, yet is highly optimized for each target

processor and its peripherals. Task schedulers are written in assembly and C data types are scaled to fit processor characteristics. C code is written to ANSI standards and optimized using C compiler extensions to specific processors.

Want to develop and test applications on a PC? Byte-BOS is fully integrated to run on a PC with DOS and can be used for your real-time PC applications or for design and testing of code for use on other processors supported by Byte-BOS.

Concerned about cost, licensing, and support? Byte-BOS is only \$1990 for source code. Executable versions can be distributed with no royalty and source code is licensed to your site for use on an unlimited number of projects. One year of technical support and revision updates are included free of charge.

Call today and get a head start on your real-time embedded or PC software application.

Byte-BOS Integrated Systems
Del Mar, CA

800-788-7288 or 619-755-8836

CIRCLE # 23 ON READER SERVICE CARD

Listing 2 A table-driven algorithm.

```
| Initialize the CRC Table for a bitwise CRC.
This procedure works for any CRC code, if
ProcessByte is adjusted as needed. |
```

```
Procedure MakeTable;
var i: Integer;
begin
  for i := 0 to 255 do begin
    CRC := 0;
    ProcessByte(i);
    Table[i] := CRC;
  end;
end;
```

```
| Send a single byte using table-
driven algorithm |
```

```
Procedure SendByte(B: Byte);
begin
  CRC := (CRC Shr 8) Xor Table[B Xor Lo(CRC)];
end;
```

obvious that we can. To see how it's possible requires one last exercise in pain.

Let's try to follow the progress of an arbitrary byte through the system. Just prior to the first shift, the LSBs of the BCC and the data byte are available at the inputs to the tap. If we define:

$$X = D \text{ Xor } Lo(C),$$

where $Lo(C)$ is the low byte of the BCC, then this value will be the LSB of X , say X_0 . As the data is shifted, X_0 will be fed into every stage that has a tap. In any given stage, all the terms in a given column are to be exclusive ORed together.

This same process takes place for every bit transmitted. The final state of the register after the eighth shift is

Listing 3 Byte-wise table generation.

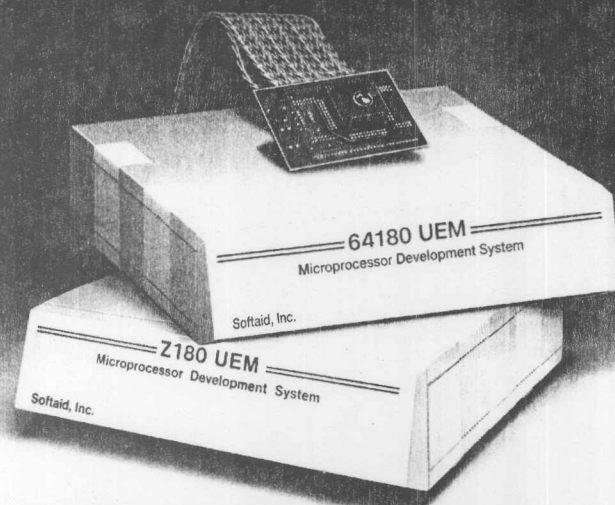
```
{ This procedure creates
the table for a byte-wise CRC. The
algorithm is specific to the CCITT
CRC. Other codes can be done in a
similar manner.
Procedure MakeTable;
var i: Integer;
    z: Byte;
begin
  for i := 0 to 255 do begin
    z := 1 Xor (i Shl 4);
    Table[i] := (z Shl 8)
      Xor (z Shl 3) Xor (z Shr 4);
  end;
end;
```

Listing 4 Byte-wise CRC without a table.

```
{ This Procedure Permits Byte-wise
CRC Without the Need for a table. The "table
entry" is built "on the fly." Note that the
algorithm is specific to the CCITT CRC.
Other codes can be done in similar manner. }

Procedure UpdateCRC(B: Byte);
begin
  B := B Xor Lo(CRC);
  B := B Xor (B Shl 4);
  CRC := (CRC Shr 8) Xor (B Shl 8) Xor
    (B Shl 3) Xor (B Shr 4);
end;
```

The 8-Bit Solution



Affordable Performance for the 64180 & Z180

Price and performance are no longer mutually exclusive. Softaid's 8-bit UEM Emulators find bugs *fast*. And only Softaid has the Fiber-Optic Link that reduces download time to virtually zero. Spend a lot less time waiting, and a lot more time debugging.

All of Softaid's UEM Emulators include built-in performance analysis and a source-level debugger for C, C++ and PL/M. Our real-time Memory Access Monitor instantly captures "stack creep" and those

programs that mysteriously "wander off." In C or assembly, the UEM automatically tracks the MMU.

Collect real-time trace data with pre, post, or center triggers. Display it in C source or intermixed with the disassembled trace data. Or, use your own clock to collect logic analyzer data.

Is there a better solution than Softaid? At \$5,495, this may be the best 8-bits of advice you ever get.

Processor	Speed
64180	10MHz
64180S	10MHz
Z180	10MHz

8300 Guilford Rd. • Columbia, MD 21046
(800) 433-8812 • FAX: (301) 381-3253

SOFTAID, Inc.

CIRCLE # 26 ON READER SERVICE CARD

shown in Figure 7 (p.44). The dotted line is to emphasize that the resulting BCC really consists of two parts:

- The high byte of the original BCC, shifted down into the low-byte position.
- The rest of the result, which depends *only* on the value of X.

Now, whatever value X has, it is, after all, only a byte, so it can only take on 256 possible values, which suggests that we can precompute the value and store it in a table. The table, of course, is constant, so it can be precomputed at initializa-

Listing 5 The XMODEM algorithm.

```
{ The XMODEM Algorithm shifts left
instead of right. This impacts most of the
procedures.
```

```
{ Bytewise, table-driven algorithm for XMODEM }
```

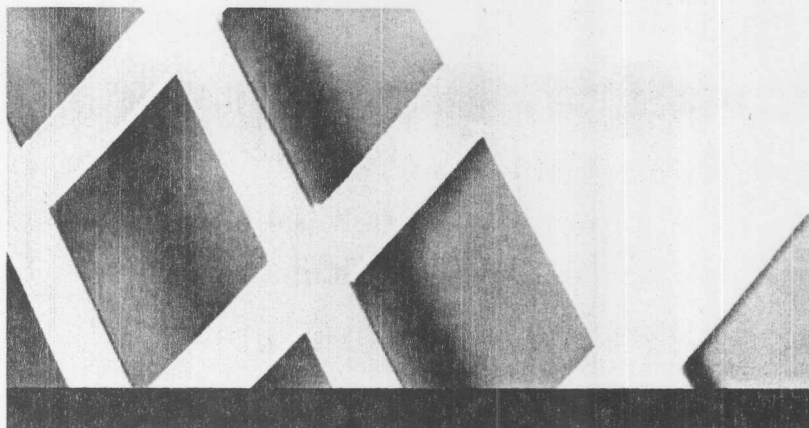
```
Procedure SendByte(B: Byte);
begin
  CRC := (CRC Shl 8) Xor Table[B Xor Hl(CRC)];
end;
```

```
{ Table Builder for XMODEM }
```

```
Procedure MakeTable;
var i: Integer;
    z: Byte;
begin
  for i := 0 to 255 do begin
    z := i Xor (i Shr 4);
    Table[i] := z Xor (z Shl 5) Xor (z Shl 12);
  end;
end;
```

```
{ Message Protocol for XMODEM. Note the two
null bytes, and the way the BCC is sent.
```

```
Procedure SendMessage(S: String);
var OldCRC: Word;
begin
  CRC := 0;
  SendString(S);
  OldCRC := CRC;
  Write('CRC Computed is ');
  WordLn(CRC);
  SendByte(OldCRC);
  SendByte(Hl(OldCRC));
end;
```



STOP WASTING TIME & MONEY

BSO/TASKING's new toolkit for developing software for the 68000 family, called **TASKTOOLS™**, beats all others on the market today. Check out these features and benefits.

C COMPILER

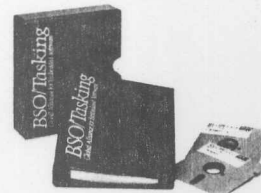
- ANSI C
- Highly optimized code
- Reentrant code
- Direct control of I/O
- Interrupt handlers may be written in C
- Floating point support
- 68040 and 68332 support

CROSS VIEW™ DEBUGGER

- All major emulators supported
- Multi-window interface
- Code & data breakpoints
- Source level tracing
- Stack tracing
- I/O simulation

ASSEMBLER & LINKER

- Motorola compatible
- FPU & MMU support
- Fully featured



TaskTools and CrossView are trademarks of BSO/TASKING.

TASKTOOLS™
The most
powerful set of
68000 software
development
tools available!

BENEFITS

TASKTOOLS™ offers you the following benefits:

- Reduced code size
- Increased productivity
- Great documentation
- Hot line support
- For PC, Sun, Vax, HP, Apollo, IBM & DEC RISC
- Multi object formats

OTHER PROCESSORS WITH SIMILAR SUPPORT

- Motorola 68HC11
- Intel 80X86
- Intel 8051 & derivatives
- TMS 320C25
- Siemens 80C166

CALL US
1-800-458-8276.
Outside U.S.A. 617-894-7800.
Fax 617-894-0551.



BSO/TASKING US
1-800-458-8276

BSO/TASKING UK
+44 252 511014

BSO/TASKING Italia
+39 2 6698 2207

BSO/TASKING Deutschland
+49 7152 22090

BSO/TASKING France
+33 1 30 54 222

Tasking LV Netherlands
+31 33 558584

CIRCLE # 28 ON READER SERVICE CARD

JANUARY 1992 EMBEDDED SYSTEMS PROGRAMMING 39

Implementing CRCs

tion time or loaded as a constant. Then, to process the CRC, all we have to do is a couple of *Xor*'s and a table lookup. The processing algorithm goes as follows:

- Compute $X = D \text{ Xor } Lo(C)$
- Use X as an index into the table, fetch $T[X]$
- Compute the new CRC as: $T[X] \text{ Xor } (C \gg 8)$

It's really quite simple. We still have to create the initial table. It turns out not to be difficult, though. Suppose that the initial BCC were zero, and we process the byte X , bitwise. Then, the result should be just what we're looking for, since all the C bits will be zero. To generate the table, all we need to do is process the integers from zero through 255, using the bitwise algorithm.

The results are shown in Listing 2 (p.34). Again, the only significant difference between the CRC algorithms is in the values used for Feedback. Now, you might think we have gone about as far as we can possibly go with the CRC algorithm, but we can still make some improvements.

The problem is that grotty bitwise table computation. We end up having to provide a bitwise *and* bitwise CRC routine, using the first one just to create the lookup table. Aesthetically, that's not so pleasing.

Even though the bit patterns of the X_i s are complex, we *can* see a pattern, and it turns out that the table entries can be generated with just a few shifts and exclusive ORs. A word of warning, though—this approach makes the Make-Table procedure algorithm-specific. The bit patterns will differ depending upon the polynomial used. The code shown in Listing 3 (p.37) is for the CCITT polynomial. The others are similar, and the rules are given in Table 2 (p.45).

Finally, in some cases (single-chip controllers, for example), speed is not as

Suppose that the initial BCC were zero, and we process byte X , bitwise. The result should be what we're looking for, since all the C bits will be zero.

important as program size. Our 512-byte table may be too much for such

Figure 6
Hand check of CRC.

Input Message = 'M' = 4Dh = 0100110b

Reverse the bit pattern, and append 16 zeros to get the dividend:
1011 0010 0000 0000 0000 0000

Generating Polynomial = $x^{16} + x^{12} + x^5 + 1 = 1\ 0001\ 0000\ 0010\ 0001$ (divisor)

Divide using bitwise exclusive or:

Quotient (discarded) --> 1011 1001

1 0001 0000 0010 0001	)	1011 0010 0000 0000 0000 0000
		1000 1000 0001 0000 1
		11 1010 0001 0000 100
		10 0010 0000 0100 001
		1 1000 0001 0100 1010
		1 0001 0000 0010 0001
		1001 0001 0110 1011 0
		1000 1000 0001 0000 1
		1 1001 0111 1011 1000
		1 0001 0000 0010 0001

Remainder (BCC) 1000 0111 1001 1001

Reverse bit pattern again to get BCC in its usual form:

BCC = 1001 1001 1110 1000 = 99E8h.

systems.

In this case, we can still get the benefits of the bitwise approach, by combining the table-build and table-lookup steps. In effect, we are building each table entry on the fly, each time we need it. The code for this procedure is shown in Listing 4 (p.37).

DETAILS AND GOTCHAS

That pretty much wraps up the basic concepts of CRC processing. However, we still have some minor nits to pick, which can turn out to be major roadblocks if you're caught unawares.

First of all, almost everyone who implements the CRC algorithms follows the rules, but every once in awhile you run across one that doesn't. We've always shown the shift registers shifting right, and corresponding to the low-bit-first rule. But it doesn't have to be done

Sim

Featu

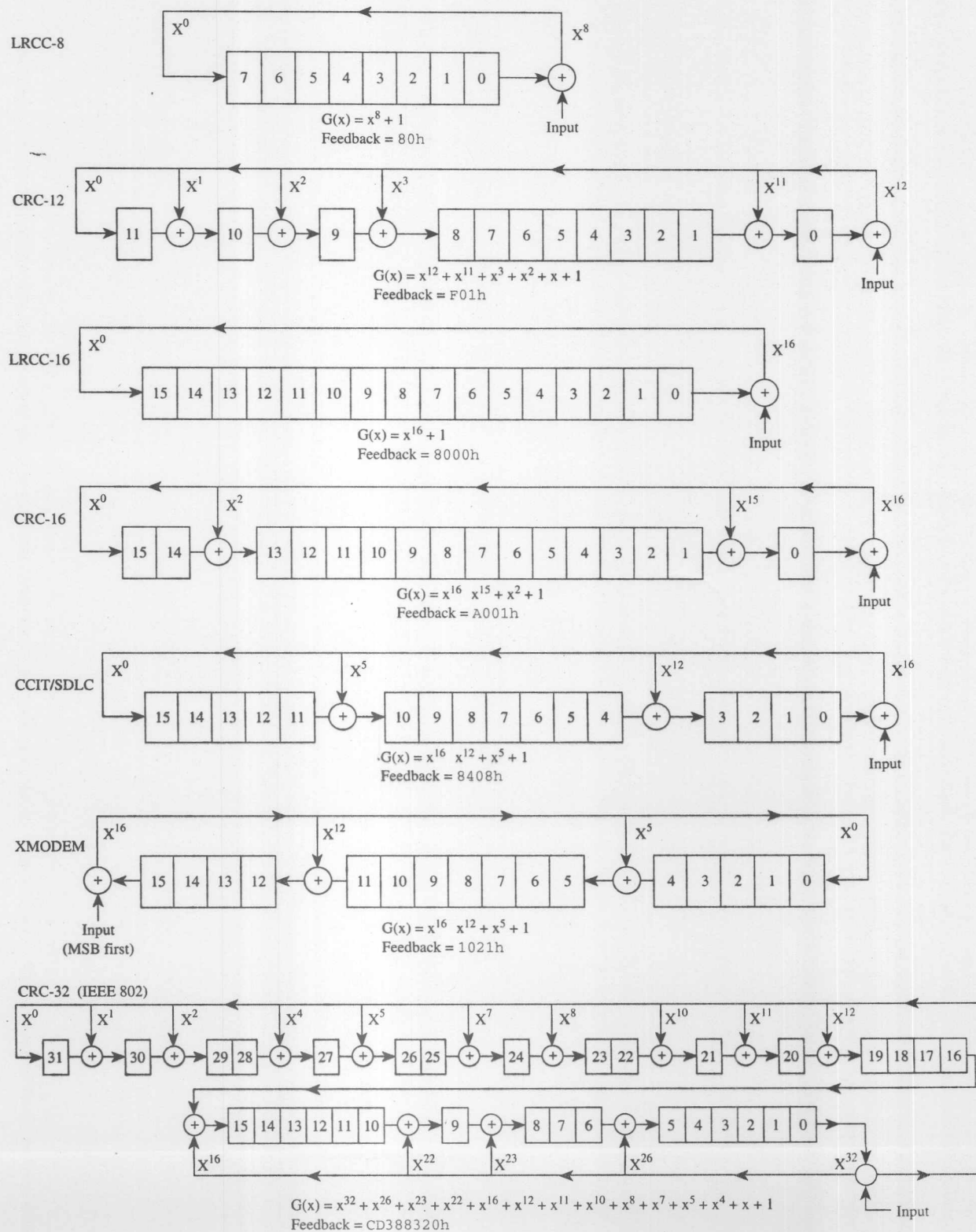
- ☐ Prior and
- ☐ Unlin res
- ☐ Fixed
- ☐ Time
- ☐ ROM
- ☐ No r
- ☐ MS-I

Proc

80x86s,
Am2900
MIPS, T

(800) 46

Figure 5
CRC standards.



Implementing CRCs

that way. The CRC will still work, whether the data is sent LSB first, first byte first, or not. It's just that nobody will be able to read the message!

The most notable violator is in the XMODEM/CRC algorithm. When Chuck Forsberg implemented this code, he found that on the 8080 it's a lot easier to shift registers left than right. Basically, the XMODEM algorithm is CCITT done backwards. The bytes are processed in order and, of course, sent LSB first (since LSB is controlled by hardware), but the CRC is done as though the bytes were being sent MSB first. Because the XMODEM code is so unique, the table-driven version of it is included in Listing 5 (p.39). The XMODEM al-

gorithm also has a couple of other peculiarities: two null bytes are processed after the message is sent. I suppose Forsberg was thinking of those added bits in the manual division process. They are not needed, but since he uses them, you need to also. Since they are there, the BCC will *not* come out to be zero when the transmitted BCC is processed, so the two are compared. XMODEM sends the BCC high byte first.

Another gotcha lies in the SDLC/HDLC algorithm used by IBM. It is basically the CCITT algorithm, but with a critical difference: instead of initializing the BCC to zero, in SDLC it's initialized to FFFFh, to catch any unwanted null bits preceding a message. (With the standard CCITT version, the BCC never changes from zero until the first non-zero bit is encountered.)

In SDLC, the resulting BCC is also

Figure 7

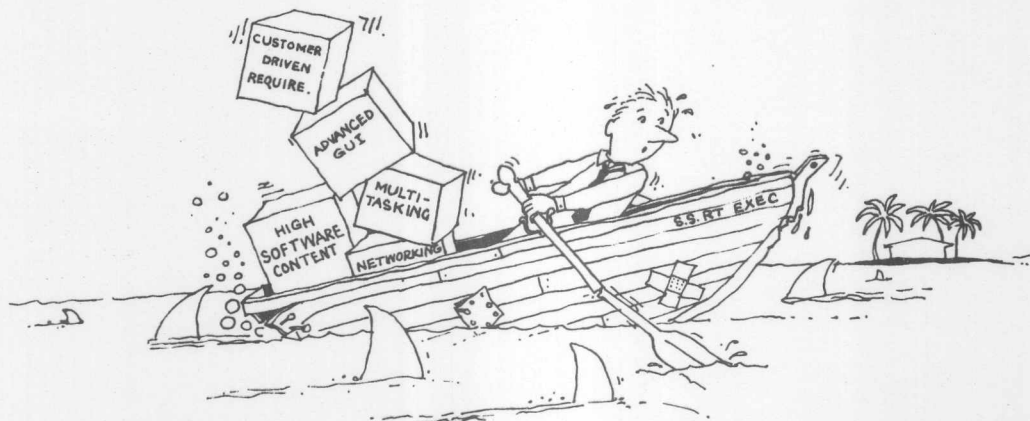
Result of processing eight data bits.

Initial value

C₁₅ C₁₄ C₁₃ C₁₂ C₁₁ C₁₀ C₉ C₈ C₇ C₆ C₅ C₄ C₃ C₂ C₁ C₀

Result after eight shifts

																C ₁₅	C ₁₄	C ₁₃	C ₁₂	C ₁₁	C ₁₀	C ₉	C ₈
																							
X ₇	X ₆	X ₅	X ₄	X ₃	X ₂	X ₁	X ₀									X ₇	X ₆	X ₅	X ₄				
								X ₇	X ₆	X ₅	X ₄	X ₃	X ₂	X ₁	X ₀								
X ₃	X ₂	X ₁	X ₀					X ₃	X ₂	X ₁	X ₀					X ₃	X ₂	X ₁	X ₀				



Your Old Patched-Up OS is Going Down VENIX Industrial Strength UNIX to the Rescue

- ◆ Licensed AT&T/USL UNIX System V 3.2/4.0
- ◆ POSIX 1003.4 Real Time Functionality
- ◆ Fast, Deterministic, Fully Preemptable Kernel
- ◆ Embedded Disk And ROM Configurations
- ◆ Homogeneous Development And Target Environment
- ◆ Build On 3,000+ Off-The-Shelf Programs

VENIX is AT&T UNIX ruggedized for acquisition and control. It's the only OS that delivers guaranteed real time, enhanced performance and *all* the UNIX development tools for 386/486 microcomputer users.

VENIX can accelerate your product development and provide the stable, open system your market demands. Call (617) 661-1230 for a FREE UNIX Technical Overview.

VenturCom
Industrial Strength UNIX

CIRCLE # 31 ON READER SERVICE CARD

converted
it's transm
sult is not
out to be a
dependent
For SDLC

One las
the tradeo
often spee
gorithms t
up with 96
is not bad
proach is
are trans
for interm
choice but
and/or re
copied int
fore it's t
proach is
the data i
the buffer
once the
can run at
link. Simi
best to re
process it

WINDING

T ha
C
fu
knowing h
tant to you
rithms we
the state o

Just re
ject with
someone t
sure that
what the a
I've seen e
comes and
plain wror
done diffe
ceiving en
error, and
hair out ti
lem. The
Good luck

Jack Cren
Tampa, F
puter prog
involved w
ment, and
did much

n the SDLC by IBM. It is algorithm, but instead of ini- in SDLC it's atch any un- a message. T version, the zero until the itered.) g BCC is also

C₁ C₀
C₉ C₈
.....
X₅ X₄
X₁ X₀

n
sition and
real time,
ment tools
ment and
ands.
Overview.

converted to its ones complement before it's transmitted. Therefore, the end result is not a zero BCC. Instead, it turns out to be a constant but non-zero value dependent only upon the polynomial. For SDLC, the magic number is F0B8h.

One last point: we've talked about the tradeoff between speed and size, and often speed is quite important. The algorithms used in XMODEM can keep up with 9600-baud transmission, which is not bad at all. However, a better approach is not to ask them to try. If you are transmitting bytes without facilities for intermediate storage, you have no choice but to process as you transmit and/or receive. But usually, the data is copied into a block buffer anyway, before it's transmitted. So, the best approach is to precalculate the BCC from the data in the buffer and append it to the buffer before you send it. That way, once the block transmission starts, it can run at the full bandwidth of the data link. Similarly, on the receiving end, it's best to receive and buffer the data, then process it between blocks.

WINDING DOWN

That about wraps up the story for CRCs. I hope you liked it. It's fun to work with them, and knowing how may turn out to be important to you someday. The bitwise algorithms we've covered are pretty much the state of the art.

Just remember: approach this subject with care. If you're working with someone to implement a CRC, make sure that all parties agree precisely what the algorithm is. Each time I think I've seen every possible variation, along comes another one—some of them just plain wrong. Remember, if anything is done differently at the sending and receiving ends, the CRC will declare an error, and you'll end up tearing your hair out trying to figure out the problem. The test-case strings should help. Good luck.

Jack Crenshaw is a consultant based in Tampa, Fla. He wrote his first computer program in 1954, and has been involved with software design, development, and methodologies ever since. He did much early work in the space pro-

In SDLC, the BCC is also converted to its ones complement before it's transmitted. Therefore, the end result is not a zero BCC.

gram and has developed numerous analysis and real-time programs. Crenshaw holds a Ph.D. in physics from Auburn University. Computer science he learned the hard way.

Table 2
Bytewise algorithms.

CCITT and SDLC/HDLC

Build Table:
z := i Xor (i Shl 4)
Table[i] := (z Shl 8) Xor (z Shl 3) Xor (z Shr 4)

Update CRC:
CRC := (CRC Shr 8) Xor Table[Data Xor Lo(CRC)];

XMODEM

Build Table:
z := i Xor (i Shr 4)
Table[i] := z Xor (z Shl 5) Xor (z Shl 12);

Update CRC:
CRC := (CRC Shl 8) Xor Table[Data Xor Hi(CRC)];

CRC-16

Build Table:
if Parity(i) then
T := \$C001
else
T := 0;
Table[i] := T Xor (i Shl 6) Xor (i Shl 7);

Update CRC:
CRC := (CRC Shr 8) Xor Table[Data Xor Lo(CRC)];

REFERENCES

Grappel, Robert. "Generating CRCs with Software," *EDN*, Oct. 1984, p. 205.
Hamilton, Dennis E. "BCCITT.CL," Penfield, N.Y.: MSW Company, 1985.
McNamara, John E. *Technical Aspects of Data Communications*. Bedford, Mass.: Digital Press, 1982, pp. 110-122.
Morse, Greg. "Calculating CRCs by Bits and Bytes," *Byte*, Sept. 1986, pp. 115-123.
Perez, Aram. "Byte-wise CRC Calculations," *IEEE Micro*, June 1983, pp. 40-50.
Peterson, W. "Cyclic Codes for Error Detection," *Proc. IRE*, Jan. 1961, pp. 228-235.
Ritter, Terry "The Great CRC Mystery," *Dr. Dobbs's Journal*, Feb. 1986, pp. 26-84.
Tanenbaum, Andrew S. *Computer Networks*. Englewood Cliffs, N.J.: Prentice-Hall, 1981, pp. 128-133.
Wecker, S. "A Table-Lookup Algorithm for Software Computation of Cyclic Redundancy Check (CRC)," Digital Equipment Corporation Memorandum, 1974.

The author would also like to acknowledge the assistance on CompuServe of Tom Catrall, Philippe Auphelle, Irv Hoff, and Dennis Hamilton.